A Typed Higher-Order Programming Language Based on the Pi-Calculus

AUTHOR(S):

Pierce, Benjamin C.; Remy, Didier; Turner, David N.

CITATION:

Pierce, Benjamin C. ...[et al]. A Typed Higher-Order Programming Language Based on the Pi-Calculus. 数理解析研究所講究録 1993, 851: 46-60

ISSUE DATE:

1993-10

URL:

http://hdl.handle.net/2433/83705

数理解析研究所講究録
第851巻 1993年 46-60

A Typed Higher-Order Programming Language Based on the Pi-Calculus

Benjamin C. Pierce
LFCS, Edinburgh

Didier Rémy
INRIA-Roquencourt

David N. Turner
LFCS, Edinburgh

July 17, 1993

Abstract

The π-calculus offers an attractive basis for concurrent programming languages. It is small, elegant, and well understood, and it supports, via simple encodings,a wide range of high-level constructs such as structured data, higher-order programming, concurrent control structures, and objects. Moreover, familiar type systems for the λ-calculus have direct counterparts in the π-calculus, yielding strong, static typing for high-level languages defined in this way.

1 Introduction

Though it originated some years before computer science itself, the λ-calculus has come to be regarded as a *canonical* calculus capturing the notion of sequential computation in an elegant, mathematically tractable presentation. Many of the fundamental issues of sequential programming languages can profitably be studied by considering them in the more abstract setting of the λ-calculus. Conversely, the λ-calculus has strongly influenced the design of many programming languages, notably McCarthy's LISP [McC78].

Milner, Parrow, and Walker's π-calculus [MPW92, Mil91] represents a step towards a canonical calculus for concurrent computation. It is a synthesis and generalization of many years of work on CCS and its relatives [Mil80, Mil89, etc.]. In the concurrency community, the π-calculus and similar "modern process calculi" are being studied aggressively, and a substantial body of theoretical results has already been accumulated. The π-calculus in its present form cannot be described as a canonical calculus for concurrent computation. But what is more important for present purposes, though more difficult to quantify, is the fact that the π-calculus appears to be a much more *computationally complete* model of real-world concurrent programs than previous formal theories of concurrency.

For example, CCS can be extended to include communication of structured data, but even so, it still lacks the ability to perform higher-order programming (the ability to send processes along channels). Thus, although there are several programming languages whose facilities for communication are based on CCS, we cannot design a higher-order language using *only* CCS as its formal foundation. The π-calculus, on the other hand, does support higher-order programming. The ability to pass channels as values between processes — its defining characteristic — turns out to yield sufficient power to construct dynamically evolving communication topologies and to express a broad range of higher-level constructs. Basic structured data such as numbers, queues, and trees can be encoded as processes using techniques reminiscent of Church's encodings in the λ-calculus. Indeed, the λ-calculus itself can be encoded fairly straightforwardly by considering β-reduction as a kind of communication. Thus, the distance between the π-calculus and a high-level notation suitable for general-purpose concurrent programming should be of the same order of magnitude as the step from λ-calculus to early dialects of LISP.

We have reached a point where a foundational calculus is sufficiently powerful to begin experimenting with higher-level designs. We can now ask the question: "If LISP (or ML) is a language based directly on the λ-calculus, then what might a language based directly on the π-calculus look like?"

2 The Pure Pi-Calculus

The active agents of the π-calculus are *processes*, which exchange information over *channels*. The most interesting process constructors are the *input* and *output prefixes* `x?y` and `x!v`. A process of the form `x!v P` outputs the value `v` along the channel `x` and then continues as `P`. This communication is *synchronous*: `P` is prevented from executing until the communication on `x` has completed. Symmetrically, the process `x?y Q` waits to receive a value along `x`, continuing as `Q` where the value received is substituted for the formal parameter `y`.

The composite process expression

```
x!v P  |  x?y Q
```

denotes `x!v P` and `x?y Q` running in parallel. Since both an input and an output on `x` are enabled, the two subprocesses may interact, and the whole system evolves to

```
--> P  |  Q{v/y}
```

where `-->` denotes the interaction and `Q{v/y}` is the capture-avoiding substitution of `v` for `y` in `Q`.

This style of *synchronous rendez-vous* on channels is well known from numerous concurrent programming languages [oD80, INM84, Hol83, Car86, Rep91, Mat91, GMP89, Ram90, Ber93, etc.] and earlier process calculi [Mil80, Mil89, etc.]. However, unlike its predecessors, the π-calculus's channels not only provide the means of communication, but are also the values exchanged during communication. This is the source of the π-calculus's surprising flexibility. In the following example we send the channel `v` along `x`,

```
    x!v P  |  v?w Q  |  x?y y!z R
--> P  |  v?w Q  |  v!z R{v/y}
```

enabling the rightmost process to use `v` to send the value `z` to the process in the middle:

```
--> P  |  Q{z/w}  |  R{v/y}
```

New channels are created using the declaration form `let new x in P end`. In the following example, this enables us to create a new channel `v`, initially private to the two processes on the left:

```
let new v in x!v P | v?w Q end  |  x?y y!z R
```

Informally, the `new` construct is evaluated by replacing the bound name `v` with a globally unique channel constant `k`, yielding the composite process

```
--> x!k P{k/v}  |  k?w Q{k/v}  |  x?y y!z R
```

which, as before, reduces to

```
--> P{k/v}  |  k?w Q{k/v}  |  k!z R{k/y}.
```

In the end, all three subprocesses in the system have access to the channel `k`. The semantics of the `new` construct guarantees that `k` is distinct from any other channels in the system; in particular, when `k` is transmitted to `R` it cannot be confused with any channels already accessible to `R`.

For completeness, we provide a constant process `skip` that does nothing at all. When `skip` appears at the end of an input or output, it is usually omitted: `x!y skip` is written just `x!y`.

We also use the `let` declaration form to write recursive process definitions. Recursive definitions are essential, since they give us the ability to express repetitive, possibly infinite, behavior. The ones we provide are very similar to those used in CCS and early presentations of the π-calculus. The following example defines a recursive process parameterized on a channel `a`. When invoked, it outputs `v` along whatever channel is provided as its argument and then calls itself using the process application form `X(a)`. One copy of `X` is then started, supplying `w` as argument. The overall behavior of the example is to output the channel `v` along `w` infinitely often.

```
let
  def X(a) = a!v X(a)
in
  X(w)
end
```

Mutually recursive definitions are also allowed:

```
let
  def X a = ...X(v)...Y(w)...
  and Y a = ..X(a)..Y(b).....
in
  X(w)
end
```

We allow a simple version of non-deterministic choice, often called *guarded choice*: an expression of the form **select P or Q end** can behave like either P or Q, where P and Q must both begin with either input or output prefixes. For example,

```
select x!y P or x!z Q end
```

can transmit either y or z along x, becoming P in the first case and Q in the second. In general, a **select** may specify any number of alternative communications: when one occurs, all the other branches of the **select** are discarded. We consider the simple input and output prefixes **x?y P** and **x!v P** to be abbreviations for the unary selection forms **select x?y P end** and **select x!v P end**.

This completes our summary of the pure π-calculus. Our description has intentionally been kept rather informal, both to emphasize its operational aspects and to avoid unnecessary formal detail in this survey paper. More rigorous accounts of π-calculus semantics can be found in [MPW92, Mil91, Mil90]. Also, the higher-level notation we describe in the following sections should not be taken as an actual programming language in its own right: we have chosen a small set of features that can be used together to tell a coherent story illustrating the power and elegance of a "bottom-up" approach to concurrent language design. A full-scale language design based on this core is underway. An early version, described in [Pie93], is available electronically with a prototype implementation. Our current implementation efforts emphasize portability, simulating concurrency on uniprocessors. The efficient implementation of π-calculus programs on distributed architectures poses significant compilation challenges.

3 Values

At this point, we find ourselves in approximately the same position as a programmer using the pure, untyped λ-calculus. In principle, everything we need for writing any conceivable program can be built up from the elements at hand: data structures, higher-order programs, and even the λ-calculus can be encoded in the pure π-calculus [Mil91, San92, Mil90]. For example, an extended language where a whole tuple of values can be exchanged at each communication step can easily be "compiled" into the pure calculus. The communication of a tuple

```
    x![u,v] P  |  x?[y,z] Q
--> P  |  Q{u/y}{v/z}
```

is translated into an initial communication of a fresh "private name," on which the elements of the tuple are then exchanged (the channel names p and q are chosen fresh):

```
    let new p in x!p p!u p!v P end  |  x?q q?y q?z Q
--> x!k k!u k!v P  |  x?q q?y q?z Q                    (k a fresh constant)
--> k!u k!v P  |  k?y k?z Q
--> P  |  Q{u/y}{v/z}
```

The fact that p is created fresh and used only for this exchange guarantees that this sequence of values cannot become interleaved with the elements of another tuple being sent along x by some other process.

Of course, for notational convenience (and perhaps also for efficiency) we certainly want to write `x![u,v]` instead of performing this encoding each time by hand. But, as in the λ-calculus, we can gain useful intuition from such encodings: if we provide a high-level notation for tuples that is behaviorally indistinguishable from the encoding, we may rest assured that its introduction does not create internal inconsistencies in the language semantics. Moreover, much of the formal theory of such high-level features can be obtained via their translation into the more tractable lower-level language.

In fact, instead of extending the syntax of communication directly just with tuples, it is convenient to introduce a more general syntax for *values*. For the moment, a value can be either a channel or a tuple of values; on each output, a single value is transmitted. Similarly, each input receives a value; in the case where a tuple is expected, its components may be named separately using a *pattern*:

```
PREFIX  ::=  ID '!' VAL PROC
             ID '?' PAT PROC

VAL     ::=  ID
             '[' <<VAL ... ','>> ']'

PAT     ::=  ID
             '[' <<PAT ... ','>> ']'
```

(Here ID ranges over channel names, VAL over values, PAT over patterns, and PROC over process expressions, and `<<VAL ... ','>>` stands for a sequence of zero or more values separated by commas. Our meta-syntactic conventions are listed in Appendix A, along with a complete grammar for the core language.)

Tuple values and tuple patterns are used symmetrically for building and destructing tuples. The pattern `[y,z]` in the process `x?[y,z] Q` is a binder for two variables y and z, whose scope is Q. Arbitrarily nested patterns may be used: a pattern such as `[[a,b],c]` matches the output of a value like `[[u,v],w]`, and binds the variables a, b, and c in the expression that follows it.

Communication of arbitrary values can be encoded in the pure π-calculus via a translation which is not much more complicated than the one given above for tuples. More details about encoding data are given in [Mil91].

Now, having distinguished different forms of data that can be communicated between processes, we must face the possibility that a value being transmitted may not have the form expected by the receiver: a run-time type error may occur. As in the λ-calculus, we can adjoin to our untyped language a simple type system capable of detecting this sort of mistake. The crucial step here is the concept of a *channel type*, which describes the values that may be transmitted along some channel:

```
TYPE    ::=  'Chan' TYPE
             '[' <<TYPE ... ','>> ']'
```

Tuple values have types like `[T,...,T]`, while an actual channel has a type like `Chan(T)`, where T is the type of the values it carries. We make the crucial assumption that each channel carries values of a single type T, which does not vary over time; this ensures that typing is a *static* property of process expressions.

The definitions of values, patterns, and types can easily be extended to other sorts of data familiar from sequential programming languages: labeled records, integers, booleans, and so on. These are all included in the syntax and typing rules listed in the appendices. Some other useful extensions, such as subtyping, recursive types, and polymorphism, are described (in various combinations) in the literature (c.f. [Mil89, Gay93, PS93, VH93, Tur93]). Indeed, recent work on static type systems for sequential object-oriented languages (c.f. [Car88, Bru93, PT93]) indicates that a general treatment of the object-oriented features that we introduce in Section 7 might be based on a combination of all of these features.

4 Process Typing

A type system ensuring the absence of run-time errors can be built using techniques familiar from λ-calculus and functional programming languages. Since processes do not reduce to values, the basic judgement of the

system is simply $\Delta \vdash P$, which can be read, "The process P obeys the constraints on its free channels stated by the context Δ."

To avoid algorithmic issues associated with type inference, we assume that each binding occurrence of a channel name is annotated with a type:

```
PAT     ::= ID ':' TYPE
            '[' <<PAT ... ','>> ']'

DEF     ::= 'new' <<ID ':' TYPE ... ','>>
```

The rules for well-formedness of processes are straightforward. For example, the rule for output prefixes states that an expression $x!V\ P$ is well formed if x is a channel (as opposed to a tuple, integer, etc.) carrying values of type T, if furthermore the value being transmitted has type T, and if the remainder P is well formed.

$$\frac{\Delta \vdash x : \mathtt{Chan}(T) \qquad \Delta \vdash V : T \qquad \Delta \vdash P}{\Delta \vdash x!V\ P}$$

Similarly, an input $x?p\ P$ is well formed if x is a channel carrying values of type T, the form of the pattern p matches T, and P is well formed under the assumptions from Δ plus the new bindings introduced by p.

$$\frac{\Delta \vdash x : \mathtt{Chan}(T) \qquad \Delta \vdash p : \Delta', T \qquad \Delta \cup \Delta' \vdash P}{\Delta \vdash x?p\ P}$$

(Here $\Delta \cup \Delta'$ denotes concatenation of contexts, giving priority to the bindings in Δ'. The auxiliary judgement $\Delta \vdash p : \Delta', T$ determines the "type" of the pattern p and the types of the names that it binds.)

A let-expression **let** d **in** P **end**, where d is a sequence of **new** and **def** declarations, is well formed if its body P is well formed in the extended environment specified by d, which is determined using the auxiliary judgement $\Delta \vdash d : \Delta'$.

$$\frac{\Delta \vdash d : \Delta' \qquad \Delta \cup \Delta' \vdash P}{\Delta \vdash \mathtt{let}\ d\ \mathtt{in}\ P\ \mathtt{end}}$$

The process **skip** is always well formed: since it has no behavior at all, it cannot misbehave. The parallel composition of two processes is well formed if they both are — the consistency of possible communications between them is ensured by checking that both are compatible with the same context.

$$\frac{\Delta \vdash P_1 \qquad \Delta \vdash P_2}{\Delta \vdash P_1 \mid P_2}$$

Similarly, a guarded choice is well formed if each of the choices is well formed:

$$\frac{\Delta \vdash s_1 \qquad \ldots \qquad \Delta \vdash s_n}{\Delta \vdash \mathtt{select}\ s_1\ \mathtt{and}\ \ldots\ \mathtt{and}\ s_n\ \mathtt{end}}$$

Appendix B presents complete sets of rules for well-formedness, value typing, and the auxiliary judgements, covering all the constructs we have described so far as well as the higher-order features introduced in the following section.

We can prove the soundness of this typing system using the combination of a subject-reduction theorem (stating that reduction preserves well-formedness) and a simple theorem stating that a well-formed process cannot "immediately misbehave":

Theorem: If $\Delta \vdash P$ and P-->P' (P reduces to P') then $\Delta \vdash P'$.

Theorem: If $\Delta \vdash P$ then P's next step cannot result in a run-time type failure.

Formal proofs of these theorems can be found in [VH93] for a system extended with recursive types. Proofs for a system including both recursive types and polymorphism appear in [Tur93], and for a system with recursive types and subtyping in [PS93].

As in the simply typed λ-calculus, simply typed π-calculus processes have principal types [Gay93, Tur93, VH93]. This leads naturally to an equivalent of ML-style polymorphism [DM82] for the π-calculus extended with **let**. (The well-known problematic interaction between ML type inference and imperative features does not arise here because the bodies of process definitions are not evaluated until they are used.) We are also exploring strategies for partial type inference in the presence of higher-order polymorphism.

5 Higher-Order Programming

We now consider the addition of *higher-order* programming features to our language. A natural approach, which has often been followed in the past [Rep91, BMT92, Mat91, GMP89, Hol83, Car86, and many others], is to integrate concurrency primitives with the λ-calculus (or, in some cases, ML). However, the functional sublanguage tends to dominate the character of languages designed in this way. This runs contrary to the aims of the present experiment, which are to investigate high-level designs retaining the essential character of the pure π-calculus. To this end, we choose an alternative notation called the *higher-order π-calculus* [San92], which remains closer in spirit to the original system.

In the pure π-calculus, the only entities that can be passed on channels are channels. We have seen how to extend this to allow structured data, where the extension can be explained by means of a straightforward compilation into the pure π-calculus. Sangiorgi [San92] demonstrates that it is also possible to extend communication to allow *processes* to be sent along channels. Moreover, this extension can again be explained by mean of a straightforward compilation into the pure π-calculus. To accomplish this without notational confusion, it is important to distinguish two fundamental notions:

- *processes*, which are ready to run with no additional outside stimulus, and
- *process abstractions*, which cannot run until they are *triggered* by some other process.

A process abstraction is a value formed by prefixing a process expression with a pattern (the keywords **abs** and = resolve any parsing ambiguities):

```
VAL     ::=  ...
             'abs' PAT '=' PROC
```

A process abstraction is triggered by *applying* it to a value:

```
PROC    ::=  ...
             ID VAL
```

For example, we can defer the evaluation of `P` by abstracting it on the empty tuple `[]`. The resulting abstraction, `abs [] = P`, can then be sent along the channel `x` and activated by applying it to the empty tuple:

```
    x!(abs [] = P) Q  |  x?y y[]
--> Q  |  (abs [] = P)[]
--> Q  |  P
```

Abstracting a process on a non-trivial pattern allows us to create parameterized process abstractions. The following example sends along `x` a process abstraction which is parametric in `a` and `b`. The receiver of such an abstraction may activate it with different arguments by applying it to different values:

```
   x!(abs [a,b] = P) Q  |  x?y (y[v1,v2] | y[v3,v4])
--> Q  |  (abs [a,b] = P)[v1,v2]  |  (abs[a,b] = P)[v3,v4]
--> Q  |  P{v1/a}{v2/b}  |  P{v3/a}{v4/b}
```

Process abstractions and applications can be translated straightforwardly into our low-level language. For example, the compilation of the previous example is:

```
  let
    new trigger
    def Server [] = trigger?[a,b] (P | Server[])
  in
    Server[] | x!trigger Q
  end
| x?y (y![v1,v2] | y![v3,v4])
```

Instead of sending a process abstraction along `x`, we send a fresh channel named `trigger`. Then, since we are representing process abstractions by channels, we replace the application of `y` to a value by an output prefix which sends the value along `y`. Thus, the applications `y[v1,v2]` and `y[v3,v4]` are compiled to the output prefixes `y![v1,v2]` and `y![v3,v4]` respectively. The compilation also creates a `Server` process that, upon receipt of a value along the `trigger` channel, activates the process `P`, thus mimicking the result of applying a process abstraction. The server requires a recursive definition, since it must be able to handle any number of outputs on the `trigger` channel.

Abstractions and process definitions are very similar constructs. Indeed, they can be unified, allowing recursively defined processes to be used as higher-order values. (Compare Standard ML [MTH90], in which recursively defined functions and anonymous function abstractions are treated uniformly.)

The compilation of abstractions and applications guides the formulation of their typing rules. We can either extend the set of types with abstraction types, `Abs(T)`, or we can officially view abstractions in terms of their encodings, conveniently confusing application with output. For brevity, we choose the latter approach here; in a complete programming language design, it is not clear which is preferable.

6 Returning Results

Assuming the usual infix operations on integers, we can write the factorial function using our programming language:

```
def Fact [x: Int, res: Chan(Int)] =
  if x == 0 then res!1 else
    let new r : Chan(Int) in
      Fact[x - 1, r] | r?i res!(x * i)
    end
  end
```

This definition of `Fact` expects two arguments: `x`, the number we wish to take the factorial of, and `res`, the channel on which to send the final result. The problem we encounter is that whenever we wish to calculate some intermediate value such as the factorial of `x-1`, we need to explicitly create a temporary channel to receive the result (here, `r`). Then we have to explicitly wait for the result (the input prefix `r?i`). In larger programs this quickly becomes cumbersome.

ML and other functional languages have demonstrated the convenience of positional notation in exactly this sort of situation. We therefore introduce abbreviations which allow us to make use of positional notation. We introduce the type abbreviation `T -> T`, which stands for `Chan[T, Chan(T)]`. For example, we can now give `Fact` the type `Int -> Int`. More importantly, we extend our syntax of values with applications:

```
VAL     ::=  ...
             VAL VAL
```

The typing rule for application is exactly as we would expect, using the abbreviation `T -> T`:

$$\frac{\Delta \vdash V : T' \text{->} T \qquad \Delta \vdash V' : T'}{\Delta \vdash V\, V' : T}$$

Using positional notation, we can now rewrite our factorial example much more concisely:

```
def Fact [x: Int, res: Chan(Int)] =
  if x == 0 then res!1 else res!(x * Fact(x - 1)) end
```

It is important to note that this result-passing convention only provides special syntax to abbreviate the *use* of processes that return results: there is no special syntax for *defining* processes that return results. The new definition of `Fact` has the same formal parameters as before, including the `res` channel. The programmer still has to explicitly return a result on `res`.

One advantage of this approach is that a process that needs to return a result can pass the responsibility onto another process. This is reminiscent of *continuation-passing style* in the λ-calculus, and also of the

delegation operation in some concurrent object-oriented languages. For example, the process `Y` below is expected to return an integer result on `res`. `X` also returns integer results on `r`. Thus, `Y` can make `X` responsible for returning its result by passing the `res` channel on to `X`. This enables `Y` to continue with some other work, knowing that `X` will deal with sending the result on `res`.

```
def X [x: Int, r: Chan(Int)] = ... r!v ...

def Y [a,b,c: Int, res: Chan(Int)] = X[a+b*c,res] | ...
```

This part of the π-calculus language involves a number of quite subtle design tradeoffs and is the subject of ongoing research. In particular, there are some subtle problems with sequentiality. It is simpler to see these in terms of the original definition of `Fact`, since there we see `Fact`'s synchronization behavior written explicitly. The crucial decision that we have made is that (in the case where `x` is non-zero) we create a new process `Fact[x - 1, r]` that runs in parallel with the process `r?i res!(x * i)`. However, the process `r?i res!(x * i)` cannot execute until `Fact[x - 1, r]` sends a result along `r`, so we actually have a sequential implementation of the factorial function. For more complicated functions, there might well be useful work which we could be doing in *parallel* with the computation of a value. One possible solution to these problems with sequentiality would be to base our high-level language on the more flexible *synchronous π-calculus* [Mil93], refining the above result-passing scheme to support a mechanism similar to Multilisp *futures* [Hal85].

7 Concurrent Objects

Another, more speculative, extension is high-level syntax for grouping related services so that they can be manipulated as a unit. In this section we sketch the essential ideas behind this extension. A full design is in progress.

A *server process* is one that communicates with its clients via some fixed set of *request channels*; for example, a buffer with one channel for inserting elements and one for removing them is a server. The only difference between a server and an arbitrary process is that a server's set of request channels remains fixed over time. Although it may sometimes exchange messages over other, transient channels (for example, sending a reply back to a client along a channel provided as part of a request), the set of channels that clients use to initiate such exchanges is always the same.

Many concurrent programs can naturally be expressed in terms of interacting servers. However, servers are not very convenient for higher-order programming: if a client process wants to send a reference to a buffer server to another client, it must pass two parameters separately; to send a reference to a more complicated server, many more might be needed. To render higher-order programming with servers practical, we need to be able to refer to a server by a single handle.

To do this, we can interpose a simple *name server* process between clients and the original server. The name server provides a single request channel, which clients can interrogate to find the request channels of the original server. For example, the name server for a buffer process maps the labels `"get"` and `"put"` into the actual request channels for the buffer's services; in effect, it is simply a record of channels with two fields `get` and `put`. The name server's single request channel forms a convenient handle for the buffer.

A server together with an associated name server form a natural and very simple notion of *object*. To build and manipulate such objects, we extend our existing syntactic categories with some new high-level constructs. First, we add a type constructor `Object`:

```
TYPE    ::= ...
            'Object' '(' <<ID ':' TYPE ... ','>> ')'
```

(The syntax and typing rules in the appendices omit the forms introduced in this section. For present purposes, it suffices to interpret `Object(x1:T1,x2:T2)` as the record type `{x1:T1,x2:T2}`.)

For example, the type

```
type IntList = Object(
                 empty: [] -> Bool,
                 fold: [[Int,Int]->Int, Int] -> Int
               )
```

says that integer-list objects have request channels named **empty** and **fold** with the given types. Clients can use **empty** to find out whether a given list is empty; **fold** is used to apply a given binary function to all the elements of the list, beginning with a given initial value (folding + with initial value 100 along a list with elements 1, 2, and 3 yields 106).

Objects are constructed using the declaration form **object**:

```
DEC     ::=  ...
             'object' ID 'with' <<ID ':' TYPE ... ','>> 'be' PROC 'end'
```

For example, an empty integer-list object is declared by:

```
object nil with
  empty: []->Bool,
  fold: [[Int,Int]->Int, Int]->Int
be
  let ready[] =
    select empty?[[],r:Chan(Bool)]
             r!true | ready[]
    or     fold?[[f:[Int,Int]->Int, a:Int], r:Chan(Int)]
             r!a | ready[]
    end
  in ready[] end
end
```

This declaration binds the channel name **nil** to the record {**empty=empty,fold=fold**}, where **empty** and **fold** are newly created channels, local to the body of the declaration. The object's behavior is specified by the body (the **let** expression), which repeatedly accepts requests on either **empty** or **fold**.

A client process invokes a service of an object by first communicating with the name server to find the appropriate request channel and then sending the actual request. This sequence of three communications (send-receive-send) is abbreviated using the following constructs:

```
PREFIX  ::=  ...
             ID '<-' ID VAL [PROC]

VAL     ::=  ...
             ID '<-' ID VAL
```

(As usual, the **PREFIX** case is for requests that do not return results and **VAL** is for those that do.) For example,

```
if nil<-empty[] then ... else ... end
```

sends the request **empty** (with no arguments) to the object **nil** and switches on the result. Nonempty lists can now be built with a **cons** operation:

```
def cons [[h:Int, t:IntList], res:Chan(IntList)] =
  let
    object l with
      empty:[]->Bool,
      fold: [[Int,Int]->Int, Int]->Int
    be
      let ready[] =
        select empty?[[],r:Chan(Bool)]
                 r!false | ready[]
        or     fold?[[f:[Int,Int]->Int, a:Int], r:Chan(Int)]
                 t<-fold [f,f[h,a],r] | ready[]
        end
      in ready[] end
    end
  in
    res!l
  end
```

Given an integer `h` and an existing list `t`, `cons` build a new object `l` and sends it back on the result channel `res`. As before, the new object responds repeatedly to `empty` and `fold` requests. It returns `false` in response to `empty`, and responds to `fold[[f,a],r]` by applying `f` to `[h,a]` and invoking the `fold` service of `t`, passing `f[h,a]` as initial value and its own result channel `r` as the result channel of the recursive call. Another request can be processed while the `fold` operation on `t` is still in progress.

This simple account of objects and their types leaves out several important features, most importantly subtyping and the definition of recursive object types. A full treatment based on recent type systems for sequential objects [Car88, Bru93, PT93] is in progress. We have also chosen to omit any mechanism for *inheritance* for the time being; its status in concurrent object-oriented languages is still a matter of controversy.

A Syntax Summary

We rely on the following meta-syntactic conventions: The possible forms of each production are listed on successive lines. Nonterminals are written in uppercase; keywords and symbolic constants appear inside quotes. An expression of the form **<<X Y Z ... 'sep'>>** indicates a list of zero or more occurrences of the sequence **X Y Z**, separated (in case there are two or more occurrences) by **sep**. For example, the production

```
'new' <<ID ':' TYPE ... ','>>
```

allows the following examples:

```
new
new x:[]
new x:[], y:[]
```

The syntactic forms of our core language are as follows:

```
PROC    ::= PROC '|' PROC
            'skip'
            'let' DEC 'in' PROC 'end'
            'select' <<PREFIX ... 'or'>> 'end'
            ID VAL
            'if' VAL 'then' PROC 'else' PROC 'end'

PREFIX  ::= ID '!' VAL PROC
            ID '?' PAT PROC

DEC     ::= 'new' <<ID ':' TYPE ... ','>>
            'def' <<ID ABS ... 'and'>>
            'local' DEC 'in' DEC 'end'
            DEC DEC
            'val' PAT '=' VAL

VAL     ::= ID
            '[' <<VAL ... ','>> ']'
            '{' <<ID '=' VAL ... ','>> '}'
            VAL '.' ID
            INT
            BOOL
            'abs' ABS
            VAL VAL
            '(' VAL ')'

PAT     ::= ID ':' TYPE
            '[' <<PAT ... ','>> ']'
            '{' <<ID '=' PAT ... ','>> '}'

TYPE    ::= 'Chan' TYPE
            '[' <<TYPE ... ','>> ']'
            '{' <<ID ':' TYPE ... ','>> '}'
            'Int'
            'Bool'

ABS     ::= PAT '=' PROC
```

We adopt the standard convention that two processes are considered identical if they differ only by a renaming of bound variables. We also require that the set of variables bound in either a pattern, mutually recursive definition, or **new** declaration must be distinct.

B Summary of Typing Rules

B.1 Processes

$$\frac{\Delta \vdash P_1 \qquad \Delta \vdash P_2}{\Delta \vdash P_1 \mid P_2}$$

$$\Delta \vdash \texttt{skip}$$

$$\frac{\Delta \vdash s_1 \qquad \ldots \qquad \Delta \vdash s_n}{\Delta \vdash \texttt{select } s_1 \texttt{ and } \ldots \texttt{ and } s_n \texttt{ end}}$$

$$\frac{\Delta \vdash d : \Delta' \qquad \Delta \cup \Delta' \vdash P}{\Delta \vdash \texttt{let } d \texttt{ in } P \texttt{ end}}$$

$$\frac{\Delta \vdash V : \texttt{Bool} \qquad \Delta \vdash P_1 \qquad \Delta \vdash P_2}{\Delta \vdash \texttt{if } V \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}}$$

$$\frac{\Delta \vdash x : \texttt{Chan}(T) \qquad \Delta \vdash V : T}{\Delta \vdash x\,V}$$

B.2 Prefixes

$$\frac{\Delta \vdash x : \texttt{Chan}(T) \qquad \Delta \vdash p : \Delta', T \qquad \Delta \cup \Delta' \vdash P}{\Delta \vdash x?p\ P}$$

$$\frac{\Delta \vdash x : \texttt{Chan}(T) \qquad \Delta \vdash V : T \qquad \Delta \vdash P}{\Delta \vdash x!V\ P}$$

B.3 Declarations

$$\frac{\Delta \cup \{x_1 : T_1, \ldots, x_n : T_n\} \vdash a_i : T_i \qquad \text{for each } i}{\Delta \vdash \texttt{def } x_1\,a_1 \texttt{ and } \ldots \texttt{ and } x_n\,a_n : \{x_1 : T_1, \ldots, x_n : T_n\}}$$

$$\Delta \vdash \texttt{new } x_1 : \texttt{Chan}(T_1), \ldots, x_n : \texttt{Chan}(T_n) \ : \ \{x_1 : \texttt{Chan}(T_1), \ldots, x_n : \texttt{Chan}(T_n)\}$$

$$\frac{\Delta \vdash p : \Delta, T \qquad \Delta \vdash V : T}{\Delta \vdash \texttt{val } p = V : \Delta}$$

$$\frac{\Delta \vdash d_1 : \Delta_1 \qquad \Delta \cup \Delta_1 \vdash d_2 : \Delta_2}{\Delta \vdash \texttt{local } d_1 \texttt{ in } d_2 \texttt{ end} : \Delta_2}$$

$$\frac{\Delta \vdash d_1 : \Delta_1 \qquad \Delta \cup \Delta_1 \vdash d_2 : \Delta_2}{\Delta \vdash d_1\ d_2 : \Delta_1 \cup \Delta_2}$$

B.4 Values

$$\frac{\Delta \vdash V_1 : T_1 \quad \ldots \quad \Delta \vdash V_n : T_n}{\Delta \vdash [V_1, \ldots, V_n] : [T_1, \ldots, T_n]}$$

$$\frac{\Delta \vdash V_1 : T_1 \quad \ldots \quad \Delta \vdash V_n : T_n}{\Delta \vdash \{l_1 = V_1, \ldots, l_n = V_n\} : \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\frac{\Delta \vdash V : \{l_1 : T_1, \ldots, l_n : T_n\}}{\Delta \vdash V.l_i : T_i}$$

$$\frac{k \text{ a constant of type } K}{\Delta \vdash k : K}$$

$$\Delta \vdash x : \Delta(x)$$

$$\frac{\Delta \vdash a : T}{\Delta \vdash \textbf{abs } a : T}$$

$$\frac{\Delta \vdash V : T' \text{->} T \quad \Delta \vdash V' : T'}{\Delta \vdash V\, V' : T}$$

$$\frac{\Delta \vdash V : T}{\Delta \vdash (V) : T}$$

B.5 Abstractions

$$\frac{\Delta \vdash p : \Delta', T \quad \Delta \cup \Delta' \vdash P}{\Delta \vdash p = P : \textbf{Chan}(T)}$$

B.6 Patterns

$$\frac{\Delta \vdash p_1 : \Delta_1, T_1 \quad \ldots \quad \Delta \vdash p_n : \Delta_n, T_n}{\Delta \vdash [p_1, \ldots, p_n] \ : \ (\Delta_1 \cup \ldots \cup \Delta_n), [T_1, \ldots, T_n]}$$

$$\frac{\Delta \vdash p_1 : \Delta_1, T_1 \quad \ldots \quad \Delta \vdash p_n : \Delta_n, T_n}{\Delta \vdash \{l_1 = p_1, \ldots, l_n = p_n\} \ : \ (\Delta_1 \cup \ldots \cup \Delta_n), \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\Delta \vdash x{:}T \ : \ \{x : T\}, T$$

References

[Ber93] Bernard Berthomieu. Programming with behaviours in an ML framework. the syntax and semantics of LCS. Technical Report 93133, LAAS-CNRS, April 1993.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *ACM Principles of Programming Languages*, January 1992.

[Bru93] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Preliminary version in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A Symmetric Integration of Concurrent and Functional Programming. In *Theory and Practice of Software Development (TAPSOFT)*, pages 184–209. Springer, 1989. LNCS 352.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems*, volume 7(4), pages 501–538, October 1985.

[Hol83] Sören Holmström. PFL: A functional language for parallel programming, and its implementation. Programming Methodology Group, Report 7, University of Goteborg and Chalmers University of Technology, September 1983.

[INM84] INMOS Ltd. *OCCAM Programming Manual.* Prentice-Hall International, 1984.

[Mat91] David Matthews. A distributed concurrent implementation of Standard ML. Technical Report ECS-LFCS-91-174, University of Edinburgh, August 1991.

[McC78] John McCarthy. History of Lisp. In *Proceedings of the first ACM conference on History of Programming Languages*, pages 217–223, 1978. ACM Sigplan Notices, Vol. 13, No 8, August 1978.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90] Robin Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. final version in *Journal of Mathem. Structures in Computer Science* 2(2):119–141, 1992.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, University of Edinburgh, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

[Mil93] Robin Milner. Action structures for the π-calculus. Technical Report ECS–LFCS–93–264, Laboratory for Foundations of Computer Science, University of Edinburgh, May 1993.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[oD80] US Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.

[Pie93] Benjamin C. Pierce. Programming in the pi-calculus: An experiment in programming language design. Lecture notes for a course at the LFCS, University of Edinburgh. Available electronically, June 1993.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993.

[PT93] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Ram90] Norman Ramsey. Concurrent programming in ML. Technical Report CS–TR–262–90, Department of Computer Science, Princeton University, April 1990.

[Rep91] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[San92] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[Tur93] David N. Turner, 1993. Ph.D. thesis, LFCS, University of Edinburgh. In preparation.

[VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings of CONCUR '93*, July 1993. Also available as Keio University Report CS-92-004.